**Naval Research Laboratory**

**Washington, DC 20375-5320**

**NRL/MR/5540--02-8616**

# A Fault Model for Survivable Applications

John P. McDermott

*Center for High Assurance Computer Systems*
*Information Technology Division*

April 22, 2002

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704-0188*

| | | |
|---|---|---|
| 1. AGENCY USE ONLY (*Leave Blank*) | 2. REPORT DATE<br>April 22, 2002 | 3. REPORT TYPE AND DATES COVERED<br>Final |

| | |
|---|---|
| 4. TITLE AND SUBTITLE<br>A Fault Model for Survivable Applications | 5. FUNDING NUMBERS<br>PE - 9999999<br>WU - 55-7287 |
| 6. AUTHOR(S)<br>John P. McDermott | |
| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)<br>Naval Research Laboratory, Code 5540<br>4555 Overlook Avenue, SW<br>Washington, DC 20375-5320 | 8. PERFORMING ORGANIZATION REPORT NUMBER<br>NRL/MR/5540--02-8616 |
| 9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) | 10. SPONSORING/MONITORING AGENCY REPORT NUMBER |
| 11. SUPPLEMENTARY NOTES | |
| 12a. DISTRIBUTION/AVAILABILITY STATEMENT<br>Approved for public release; distribution is unlimited. | 12b. DISTRIBUTION CODE |

13. ABSTRACT (*Maximum 200 words*)

Survivability for an application is the ability of the users to complete their mission in the presence of faults (the implication is that some faults are malicious). This naturally leads to the need for precise descriptions of the faults to be survived. A survivability-oriented model of fault events should describe aspects pertinent to restoration and response. It should also classify fault events according to their impact on survivability, that is, how the damaged system continues to support its mission.

This report models a fault as a four tuple. The four tuple describes the propagation of the fault, the faulty computation it induces, the required means of repairing the fault, and the fault's impact on the mission. We use the model to describe the effect of survivability on security and identify 10 general assertions that must be true of every security mechanism in a survivable environment.

| | |
|---|---|
| 14. SUBJECT TERMS<br>Information system survivability<br>Fault tolerance | 15. NUMBER OF PAGES<br>22 |
| | 16. PRICE CODE |

| | | | |
|---|---|---|---|
| 17. SECURITY CLASSIFICATION OF REPORT<br>UNCLASSIFIED | 18. SECURITY CLASSIFICATION OF THIS PAGE<br>UNCLASSIFIED | 19. SECURITY CLASSIFICATION OF ABSTRACT<br>UNCLASSIFIED | 20. LIMITATION OF ABSTRACT<br>UL |

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std 239-18
298-102

## CONTENTS

# A FAULT MODEL FOR SURVIVABLE APPLICATIONS

## 1 Introduction

Faults adversely impact the survivability of an application system. Survivability for an application is the ability of the users to complete their mission in the presence of faults (the implication is that some faults are malicious). This definition naturally leads to the need for precise descriptions of the faults to be survived. A survivability-oriented model of fault events should describe aspects pertinent to restoration and response. It should also classify fault events according to their impact on survivability, that is, how the damaged system continues to support its mission.

This report models a fault as a four tuple. The four tuple describes the propagation of the fault, the faulty computation it induces, the required means of repairing the fault, and the fault's impact on the mission. We use this model to describe the effect of survivability on security and identify ten general assertions that must be true of every security mechanism in a survivable environment.

The purpose of this model is to understand the impact of different faults on different architectures. It is not intended as a general fault removal or fault prevention technique. It is not a substitute for assurance engineering, independent verification, or validation but rather is a means of comparing the merits and demerits of design choices applied to survivable applications. So the model can be a key building block of an assurance engineering or independent verification activity. The model serves as a framework for defining flaw taxonomies and flaw spaces for design studies.

The model's perspective is large-scale application systems. Large-scale application systems are made up of relatively complex software processes supported by an infrastructure of other software and hardware. Some of these contain obscure legacy logic of dubious quality. The effects of composing these processes has usually not been fully investigated. Large scale systems experience disruptions to services, loss of connectivity, and periods of instability. Dependability is low, for many application processes. We assume that, at any point in the overall system history, some components of large-scale systems are experiencing failure. The chief impacts of this on our model are: 1) that complex faults are likely and 2) the necessary conditions for survivable operation must be enforced in an environment where even relatively weak assertions can be invalidated.

Some of the work on survivable systems has included a notion that Byzantine faults are "arbitrary" faults. In the original work by Lamport, Shostak, and Pease [6], components exhibiting Byzantine faults send "conflicting information to different parts of the system." There is a significant distinction between "arbitrary" and "sending conflicting information to different parts of the system." The latter suits the discovery of agreement protocols for fault tolerance but the former notion includes faults not addressed by agreement protocols and is much less precise. On the other hand, the distinction is very useful in understanding survivability. In a survivability situation, we expect attacks designed by humans. It is reasonable to expect that the (best) human attackers will understand all of the research results that have been applied by the defense. These attack designers will seek to invalidate one of more of the conditions upon which survivability depends. For this

Manuscript approved March 15, 2002.

reason, we will distinguish faults as either *attacks* or *accidental faults.*

We agree with Avizienis, Laprie, and Randell [1] that Byzantine faults are faults where processes communicate inconsistent events to their peers.

### 1.1 Applications, Processes, Environments, and Infrastructures

To characterize faults with respect to survivability, designers must address the specific application of the survivable system (that is the goals the application is trying to achieve), the system's environment, and the system's infrastructure. The *environment* of a process includes both clients and peers of a process. For our purposes, a peer process is a process that is neither a client nor a server (contractor) to the excepting process, but is accessible (i.e. can be a target, see Section 2.1 below). The *infrastructure* of a process is the set of processes that provide services to it. This includes host CPU's, network connections, operating systems, etc. as infrastructure processes.

### 1.2 Process Algebra

Process algebra, typified by the algebra CSP [4], is one model of computation. It is widely understood and well-suited to describing interactions between systems of concurrent processes. This report makes extensive use of process algebra. In Sections 1, 2, and 3, the use is readily understood by the general reader. The example application of the model requires more specific knowledge of CSP. In all sections, the results are independent of process algebra per se, and could be explained with other models of computation.

## 2 A Survivability-Oriented Model of Faults

We classify fault events in terms of a four tuple: *(target, mode, damage, invalidation).* The *fault mode* describes the behavior of a process that has experienced a fault, in particular the future output events, but also other events that affect the future behavior of the process. (The Byzantine nature of a fault would be part of its mode.) The *fault target* of a fault event describes the set of processes that are affected by the fault event. Some fault modes are defined in terms of their effect on the fault target. *Fault damage* describes the effect of a fault on the representation of a process, that is, how it must be repaired as part of a restoration of services. *Fault invalidations* are the most important part of our fault model. Invalidations characterize the relationship of a fault event to an application's survivability.

### 2.1 Target

Including a target set for a fault event allows us to show how the event impacts the infrastructure and environment of the process experiencing the fault event. Each fault is an event experienced by some process $P$, the faulting process. The fault

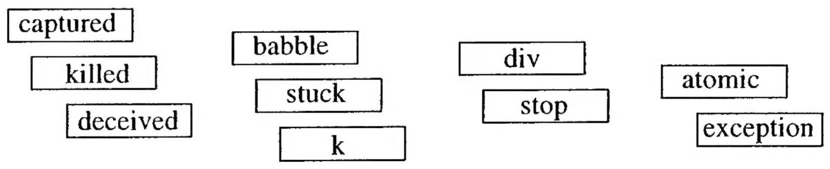

Figure 1: Fault Modes

causes $P$'s behavior to change. The changed behavior may include error events, that is, incorrect events that may cause a subsequent fault in processes that accept it. The *target of a fault* is the set of processes that have communication or memory shared with the faulting process. Notice that a process $Q$ may be in the target for a fault in process $P$, but correctly handle the error events. Finally, we point out that we need to include all potential communication channels or shared memory by which processes might become the target of a fault, and not just those that make our application work properly.

On the other hand, we should not include descriptions of the response of target components to error events. It is reasonable and even critical to describe this , but attaching it to the target draws us into model where each fault contains the closure of all faults that could depend on it. Again, this closure is useful, but better left distinct from individual fault descriptions.

## 2.2 Mode

We describe a fault mode as a parameterized event $fault(P)$. A $fault(P)$ event takes a process $P$ as a parameter. Parameter process $P$ describes the behavior of the faulting process after $fault(P)$ happens. We find it helpful to define the parameter process as a combination of elementary fault modes. The elementary fault modes are: *captured, killed, deceived, babble, stuck, k, div, stop, atomic,* and *exception.* Figure 1 shows the fault mode ordering.

The first two modes, captured and killed, represent faults that propagate error (i.e. bad) processes, by replacing good processes with bad processes. The next three modes, random, stuck, and k, represent faults that cause processes to propagate error events, but not respond to their environment. The next two modes, div and stop, represent faults that do not propagate error events, but do cause the faulting process to quit functioning. An atomic fault may cause the omission of an event or the failure of a process, but the system will be in a consistent state after the fault. Finally, an exception mode fault is a fault that is detected and handled without error.

A *captured mode* fault describes a fault event that results in a process $P$ gaining control of another process $Q$. The process $P$ now decides which events the captured (i.e. faulting) process $Q$ will participate in. A process experiencing a captured fault continues to base its future behavior on external events.

A *killed mode* fault is one where a faulting process $P$ is erroneously forced into a different state, e.g reset or reconfigured. There is no controlling process that decides the future behavior of $P$ but $P$ incorrectly interrupts its processing and resumes in an error state. (The analogy is to the Unix kill command, i.e. process $P$ receives "`kill -s SIGHUP`".) A process experiencing a killed fault continues to base its future behavior on external events.

A *deceived mode* fault describes a fault event (almost always due to an attack) that is accepted by the faulting process, but not recognized by it as exceptional. For example, if an attacking process $P$ is communicating error events $d_i$ to a set of replica processes, a deceived mode fault in one of the replica processes $Q_i$ would result when $Q_i$ accepted the error events $d_i$, but did not recognize them as errors. The distinction between deceived mode and captured or killed mode is that a process experiencing a deceived mode fault is neither controlled in arbitrary ways nor reconfigured, but only accepts and acts on incorrect events taken from its own alphabet. A process experiencing a deceived fault continues to base its future behavior on external events.

A *babble* fault happens when a faulting process starts generating externally visible events, randomly, without consideration of events presented to it by its environment. We model a babble fault as a process *Babble(M)* that takes a transition probability matrix $M$, defined over $\alpha Babble$. The alphabet $\alpha Babble$ of the babble fault process includes the incorrect events that define the specific fault behavior. The alphabet may not include successful termination (represented in CSP by the event ✓ or tick). We treat the traces of *Babble(M)* as a Markov chain. A process experiencing a babble fault no longer bases its future behavior on external events and does not resume normal operation.

A *k fault* describes an intermittent fault involving externally visible events. A k fault process is represented by a process $Fault_k(P)$ which takes another process $P$ as a parameter. The alphabet of process $P$ includes the fault events that define specific fault behavior. If process $Q$ experiences a $Fault_k(P)$ fault then $Q$'s behavior is the the sequential composition of $k$ instances of process $P$ followed by process $Q$. A process experiencing a k-fault no longer bases its future behavior on external events and does not resume normal operation, during the k repetitions of parameter process $P$.

A *stuck fault* describes a process which is generating externally visible events, in a periodic fashion. Like k fault, a stuck fault is represented by a parameterized process, in this case *Stuck(P)*, that takes another process $P$ as a parameter. A process *Stuck(P)* repeats the events of process $P$ indefinitely. A process experiencing a stuck fault no longer bases its future behavior on external events and does not resume normal operation.

A *diverge fault* causes a process to be replaced by the standard CSP process

$$div = (\mu P \bullet a \rightarrow P) \setminus a$$

The *div* process carries out an unbounded sequence of (invisible) internal events $\tau$ and thus models a process that is running but not communicating. A practical distinction between *div* and *Stop* is that *div* continues to consume resources from the infrastructure while *Stop* does not. Some network management and IDS sensors might mistakenly consider a diverge faulting process to be running properly.

A *stop fault* causes a process to act like (be replaced by) the standard CSP process *Stop*. This represents a process that has crashed (or its supporting infrastructure has crashed). A stop fault should always be detectable.

A fault that is less severe that a stop fault is an *atomic fault*. An atomic fault has no visible effects outside the faulting process, other than the non-occurrence of some events that are replaced by the atomic fault event. The process experiencing the atomic fault event is serializable [2] and at least recoverable. If it faults, it leaves no effect on the system. The idea here is that something happened that the process could not prevent, but by design no side effects were left after the fault.

The final and least severe fault behavior is an *exception fault*. An exception fault describes the occurrence of a fault that is recognized and reported by the process that experiences it. A process must have a behavior defined for each exception that it can recognize and report. Exception faults are recognized and handled, even if the handling is only to raise the exception to another process. The specific behavior of a process experiencing an exception fault is not constrained beyond the need to recognize the exception and include at least one event that constitutes reporting or handling of the exception.

None of these elementary fault modes defines a Byzantine fault by itself. An elementary captured, killed, deceived, or babble fault could also be exhibiting Byzantine behavior, or we could describe a compound fault that was also Byzantine.

## 2.3 Damage

We define four damage classes that represent the effect of a wrong input event on the representation and state data of a process that receives it. Damage classification is useful for characterizing the kind of repair that would be needed to restore a damaged process. In order of decreasing severity the damage modes are: *von Neuman damage, Harvard damage, post-condition damage,* and *pre-condition damage*

Von Neuman damage is damage to the underlying representation of the events of a process. The damage will persist until either the infrastructure or the environment repairs the representation. Furthermore, we do not expect to repair the damage by sending commands or events to the faulting process. A familiar example of von Neuman damage is a buffer overflow attack that damages (manipulates) executable code. The infrastructure of a process is responsible for preventing von Neuman damage.

Harvard damage is damage to the underlying representation of the data (local or shared) of a process. The damage will persist until either the infrastructure or the environment repairs the representation. Erasing a database file (at an inappropriate time) would be a familiar example of Harvard damage. The infrastructure of a process is responsible for preventing Harvard damage.

Post-condition damage is (self-inflicted) damage to the state data of a process. The data representation is correct, but due to a fault event the receiving process sets its state data to an incorrect value. The result is that the process begins to fault its (sequential) post-condition. An example of post-condition damage

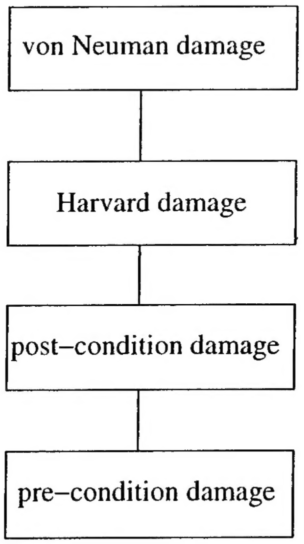

Figure 2: Damage

would be a process setting a sequence counter to the wrong value and erroneously aborting a valid run of a protocol. We normally expect some chance of repairing post-condition damage by corrective inputs to the process. A process is responsible for protecting itself from post-condition damage.

Pre-condition damage is the damage caused by a wrong event that violates the (sequential) pre-condition of the process, but does not cause any of the more serious forms of damage. Pre-condition damage describes transient faults. A process is responsible for protecting itself from pre-condition damage. A process that suffers pre-condition damage does not need to be repaired, but the cause of the error event needs to be dealt with or the damage may occur again.

## 2.4 Invalidation

The most important concept regarding a fault is the set of assertions that it invalidates. An application, its infrastructure, and its environment are organized on the basis of certain assertions. The processes of the application must enforce (implement) some of these assertions. The infrastructure or environment of a process must enforce the remaining assertions. If $A$ is an assertion, then *$P$ an enforcing process for $A$* is a process $P$ that must be working properly for $A$ to hold. The set of all enforcing processes for assertion $A$ is its enforcement set. So every assertion about a survivable system has a (possibly empty) enforcement set. Empty enforcement sets are a consequence of flawed design or specification. There should be no empty enforcement sets in a properly specified and designed application. On the other hand, most solutions make certain assumptions that they do not enforce. For example, agreement protocols assume that the participating processors do not have common mode faults where a single fault can affect all processors. In our survivability-oriented fault model, these assumptions become assertions subject to invalidation. These kinds of assertions must either be enforced by the infrastructure or the environment.

The four classes of damage we described previously constitute four implicit assertions for every process in a survivable system, namely that neither von Neuman, Harvard, post-condition, nor pre-condition damage can happen to a process. If a particular process assumes these conditions, then its infrastructure or environment must enforce the condition as an assertion, i.e., the enforcing process(es) must be in the environment or infrastructure.

Faults can cause damage that invalidates an assertion. A faulting process can kill, deceive, or capture some of the enforcing processes. An enforcing process can fail to enforce an assertion $A$ on a temporary basis, when it experiences k, atomic or exception faults. Each fault event in our model has an invalidation: the set of assertions that it invalidates.

We characterize the survivability of an application, system or architecture by identifying its assertions, their enforcement sets, and the invalidations of the fault events the system may experience. It is critical for the defenders to identify all of the assertions that impact the survivability of an application. The defense needs to be clear about the kinds of fault events that have non-empty invalidations and the assertions that have empty enforcement sets.

### 2.5 Identification of Enforcement Sets

The first step in finding the enforcement sets of an application is finding all of the pertinent assertions. Some of these will not be explicit; in fact, some of the most critical may not be explicit. For example, there may be an expectation that a group of components does not experience a common mode fault, that all faults are independent.

When we know the assertions, we can find the corresponding enforcement sets. For a given assertion $A$, we find the enforcement set by removing processes that do not enforce $A$. The remaining processes are the enforcement set for $A$.

### 2.6 The Null Fault

We can define the null fault as $(\emptyset, \mathit{fault}(\mathit{Skip}), \textit{post-condition}, \emptyset)$ which has no effect on any system component. This is useful in formal approaches that may require a fault to be present at all times.

## 3 Modelling the Effect of Faults on Security and Survivability

This fault model can address the impact of fault events on security. Since the model looks at invalidations applied to enforcement sets, we do our assessment in terms of the enforcement mechanisms. We adapt and extend a framework due to Landwehr et al. [5] that describes the requirements on logical structures for security mechanisms. The original framework was intended for host hardware mechanisms only, but, since it addresses requirements for logical structures, it is easily generalized. An enforcement mechanism should be able to

- define and separate domains
- establish an initial domain for itself
- link external actors with domains
- control inter-domain communication
- control communication with the environment
- detect and handle faults

## 3.1 Define and Separate Domains

Computation is manipulation of physical symbols. Domains limit the set of possible symbols and manipulations. In hardware, domains are made up of instructions that may be applied to bits in memory cells. Whatever the model of computation, a security mechanism built on that model must be able to limit the possible computations as necessary. A model of domain definition and separation must be concerned with the implementation of a mechanism if we are going to investigate the effect of exceptions on it. Domains must not be accomplished by specification, but by describing the actual enforcement mechanism, in order to show how the mechanism can be impacted by faults. Rushby and Randell [7] have identified four basic approaches to domain separation: physical, cryptographic, temporal, and logical. These basic approaches can be combined as in the Multiple Single-Level (MSL) approach to security, where physical and logical mechanisms are used to provide mandatory separation of domains.

## 3.2 Establish Initial Domains

A security enforcement mechanism must either protect itself from tampering or rely on a trustworthy external mechanism. Security mechanisms rely on an initial domain for self-protection. Most cryptographic solutions rely on physical and logical mechanisms to establish a separate initial domain; that is, they receive some master key from outside their own domain and this key 1(and others) are protected by unspecified means. Network security mechanisms, which may include several cryptographic components, frequently are designed to initialize themselves in domains enforced on designated trusted host systems. (Here, we would consider a cryptographic peripheral with its own processor and memory to be a "designated trusted host system.) A model of initial domain construction should address how this initial domain is implemented. This may require modelling of things normally left out, including the internal representation of processes being initialized.

A separate model of initialization, for the specific mechanism, may be a better approach.

## 3.3 Link External Actors With Domains

Accountability has always been a fundamental part of security. In the case of information systems, linkage of external actors with domains is also important for intrusion detection and automated response. Linkage can be established physically (e.g. biometrics), logically, or cryptographically. (We have been unable to come up with a plausible temporal authentication mechanism.) Notice that it is important to preserve the linkage over the life of a domain and perhaps beyond, if the mechanism is to support intrusion detection.

## 3.4 Control Inter-Domain Communication

Security mechanisms must also be able to control communication between the domains they enforce. Controlled inter-domain communication is the reason for having security mechanisms. If there was to be no inter-domain communication, then security could be achieved by isolating each process but then no information would be shared. For most systems, this would be a useless arrangement. Access control mechanisms exercise control over inter-domain communication via shared memory. Network security mechanisms exercise control over inter-domain communication via session keys and routing. When we model this aspect of a security mechanism, we should model both the means of communication and the way that this communication is controlled.

## 3.5 Control Communication With the Environment

The environment of a security mechanism is a source of commands and data with unknown provenance. A security mechanism needs to control those external communications in order to control computation. Values that are communicated to the environment are no longer under the control of the security mechanism. A security mechanism has trust relationships with its environment, accepting events and communicating values on trust. In some designs, a mechanism can use identification and authentication to screen out untrusted processes in the environment. Some mechanisms accept all events, but do not respond to events that would violate their policy. In other cases, the architecture provides the trust, i.e. all communications over a specific channel are trusted. The distinction between this requirement to control external communication and the need to link actors to domains corresponds to the distinction between prevention and detection. Control allows the security mechanism to disregard undesirable events but linkage does not. In any of these cases, a model for analyzing the effects of fault events on security must show either that there are enforcing sets that can prevent or enable communication with the general environment, or that the security mechanism is connected to its environment in a trustworthy fashion. In the latter situation, we can look at the connected part of the environment for the enforcing sets and also for unintended transitive connections to untrustworthy processes.

### 3.6 Detect and Handle Faults

Security mechanisms must be able to both detect and handle faults that impact them. Some mechanisms, like access controllers, may have fault detection capabilities for almost every kind of fault that can impact them. Other, such as cryptographic authentication protocols, may appear to have no fault detection or handling, especially when viewed at the Alice, Bob, and Yves level of abstraction. However, these mechanisms assume that faults do not allow an intruder to bypass them or interfere with their functioning. Implementations of these kinds of mechanisms should be designed to detect and handle faults reported by the mechanism infrastructure. Like survivability analysis for domains, initialization, and communication, fault detection or handling requires us to model its implementation.

### 3.7 Ten General Assertions

These six requirements also constitute implicit assertions that a security mechanism makes about itself. We combine these with the four damage assertions to get a total of ten mechanism- and policy-independent security-relevant assertions that could potentially be invalidated in any survivability environment. To asses the survivability of a specific security mechanism we would add policy- and mechanism-specific assertions to these ten universal assertions. At this point we could investigate the effect of specific faults, using the four part model.

## 4 A Process Algebra Example

An example will clarify our model and its application. Consider a simplified financial payment system, which is supposed to be survivable. Our survivable payment system has four tolerable forms of service $R$ and survives two kinds of faults from its environment; it matches the diagram of Figure 3. The services are $R_1$ *Preferred*, $R_2$ *Major Customers*, $R_3$ *Financial Markets*, and $R_4$ *Fail Stop*. The services depend on two databases $c$ and $m$ that support major customers and financial markets respectively. We propose an FRS-based architecture [3] to satisfy the specification. We assume the user workstations have no persistent user-modifiable storage and assume that suitable threshold schemes are used by the access controller to provide fragment keys. Each database is split into two fragments, then scattered and replicated across four servers *Deedee, Erica, Fritz*, and *Greg*. In practice, a relatively large number of fragments is needed, say 16, but this would make our example cumbersome. Likewise, in practice the scattering and replication is variable, but modelling the pseudo-random inter-server algorithm that decides it is also too complex for this example. In our simple model, Servers *Deedee* and *Fritz* have fragments $c_1$ and $m_1$, servers *Erica* and *Greg* have fragments $c_2$ and $m_2$. Authorization to access the databases in various modes is given by an authorization

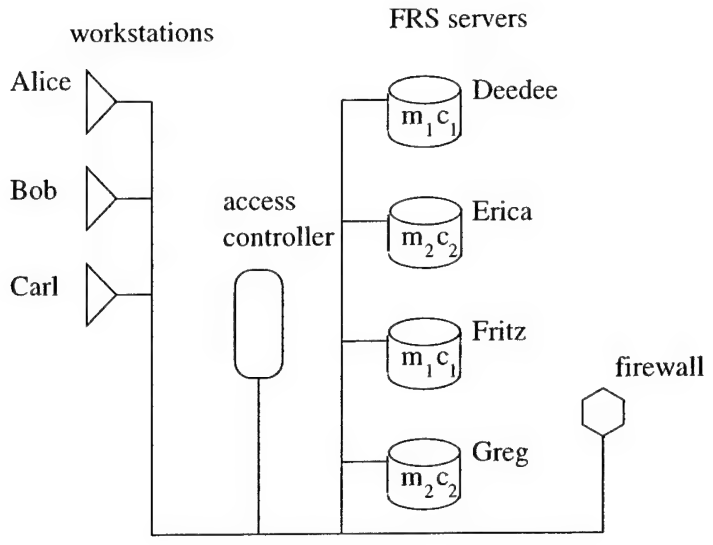

Figure 3: FRS Architecture for Financial Transactions

server. (This server is itself protected by FRS, with threshold schemes.) Our FRS architecture is shown in Figure 3.

## 4.1 Security Policy

The security policy is that: (1) Alice may access both the $c$ and $m$ databases; (2) Bob may access just the $c$ database; (3) Carl may only access the $m$ database. These three conditions constitute the application-specific assertions that we make about this example system.

## 4.2 Process Algebra Model

We will model this FRS system using CSP process algebra and specify two faults using our model. One fault will be a fault an FRS mechanism can survive and the other will be a fault it cannot survive. The fault model will show us how the FRS mechanism is affected by the faults.

The local area network is modelled as a collection of sequential communications interleaved over a compound index of network links *Links* and messages *Msgs*. The set of network links *Links* is the Cartesian product $Worksta \cup Serv \times Worksta \cup Serv$ of host names (*Worksta* for workstation names and *Serv* for server names)

$$LAN(s, d, m) = in.s?s.d.m \rightarrow Skip$$

$$LAN = \big|\big|\big|_{\substack{s.d \in Links \\ m \in Msgs}} LAN(s, d, m)$$

Each communication sends a message $m$ from a source address $s$ to a destination address $d$. The messages are compound objects, but the LAN does not look inside them. Message events are mapped to the messages sent by the workstations and the hosts, so we could have $m = write.x.f_1.s$, if the message is a write request, with sequence number $s$, to set fragment $f_1$ to $x$. Notice that process LAN has input $in.s$ and output channels $out.d$ for every link $s.d$, so workstation-to-workstation or server-to-server communication is possible, even though our well-behaved processes do not engage in such communication.

The LAN does not include the firewall because we will not be examining faults that involve the firewall. The LAN does not include the authentication server because we will not be examining faults in communication with the authentication server. We will model the authentication server's control of access as a process $TAM$ which has a local access control matrix $A$. To simplify the matters, the matrix is fixed. Each workstation will check with the authentication server before accessing a database. This models the management of fragment keys in the implementation. Since fragment keys allow all modes of access, there is only one security privilege $a$ in the set of access privileges $R$.

$$\begin{aligned} TAM = \;& tamin?w.d \in WD \rightarrow \\ & \mathbf{if}\, a \in [w, d]\, \mathbf{then}\, tamout!w.d.a\, \mathbf{else}\, tamout!w.d.\emptyset \rightarrow \\ & TAM \end{aligned}$$

The workstations are modelled by processes *Alice, Bob*, and *Carl* that act as event generators to drive the model. Each workstation process is a collection of parameterized sequential read and write transactions, interleaved over a compound index. The index for the read transactions comprises readable fragments $f \in FRS_r$, and read transaction identifiers $s \in ReadID$. An element of the readable fragments set $FRS_r$ is a quadruple $f = h_1.f_1.h_2.f_2$, where $h_i$ denotes a server and $f_j$ denotes a fragment. Set $FRS_r$ only contains useful quadruples; that is, $m_1.Deedee.m_2.Greg$ is in $FRS_r$ but $m_1.Deedee.m_2.Fritz$ and $m_1.Deedee.f_1.Fritz$ are not. We allow the read transaction to inspect the parts of fragment $f$. We use $f_i$ to denote either the name (in the outgoing request) or the value (in the return from the host) of a fragment. A single read transaction for the workstation *Alice* is

$$\begin{aligned} & Read_{Alice}(f, s) = \\ & \quad ((out!alice.h_1.read.f_1.s \rightarrow in?h_1.alice.f_1.s \rightarrow Skip) \\ & \quad ||| \, (out!alice.h_2.read.f_2.s \rightarrow in?h_2.alice.f_2.s \rightarrow Skip)) \\ & \quad || \, (out!alice.h_1.read.f_1.s \rightarrow out!alice.h_2.read.f_2.s \rightarrow Skip) \end{aligned}$$

(The second parallel process is an auxiliary process used to force the fragment reads to occur in a specific order.) We precede the basic read process with a security check

$$\begin{aligned} & CheckedRead_{Alice}(f, s) = \\ & \quad tamin!alice.f \rightarrow tamout?alice.f.p \rightarrow \mathbf{if}\, p = a\, \mathbf{then}\, Read_{Alice}(f, s) \end{aligned}$$

Now we can interleave all of the reads from workstation Alice

$$Reads_{Alice} = \interleave_{\substack{f \in FRS_r \\ s \in ReadID}} CheckedRead_{Alice}(f, s)$$

The index of a single write transaction comprises new fragment values $x \in Values$, fragment replicas that must be written $f \in FRS_w$, and write transaction identifiers $s \in WriteID$. An element $f$ of $FRS_w$ is an octuple $h_1.f_1.h_2.f_2.h_3.f_1.h_4.f_2$ (with fragments $f_i$ and servers $h_j$) that describes a meaningful combination of fragment replicas and host servers. We allow the write transaction to inspect the parts of fragment replica $f$. We also assume, to simplify, that the hosts have a way of mapping value $x$ to the fragments they hold. A write transaction for workstation process Alice is

$$\begin{aligned}
&Write_{Alice}(x, f, s) = \\
&\quad ((out!alice.h_1.write.x.f_1.s \to in?h_1.alice.ack.s \to Skip) \interleave \\
&\quad\ (out!alice.h_2.write.x.f_2.s \to in?h_2.alice.ack.s \to Skip) \interleave \\
&\quad\ (out!alice.h_3.write.x.f_1.s \to in?h_3.alice.ack.s \to Skip) \interleave \\
&\quad\ (out!alice.h_4.write.x.f_2.s \to in?h_4.alice.ack.s \to Skip)) \\
&\quad \| \\
&\quad (out!alice.h_1.write.x.f_1.s \to out!alice.h_2.write.x.f_2.s \to \\
&\quad\ out!alice.h_3.write.x.f_1.s \to out!alice.h_4.write.x.f_2.s \to Skip)
\end{aligned}$$

Like the read transactions, there must be a security check first

$$\begin{aligned}
&CheckedWrite_{Alice}(x, f, s) = \\
&\quad tamin!alice.f \to tamout?p : R \to \mathbf{if}\ p = a\ \mathbf{then}\ Write_{Alice}(x, f, s)
\end{aligned}$$

Now we can interleave all of the writes from workstation process Alice

$$Writes_{Alice} = \interleave_{\substack{x \in Values \\ f \in FRS_w \\ s \in WriteID}} CheckedWrite_{Alice}(x, f, s)$$

We get the workstation process *Alice* by interleaving $Reads_{Alice}$ and $Writes_{Alice}$ as

$$Alice = Reads_{Alice} \interleave Writes_{Alice}$$

We combine all of the workstation processes by interleaving to get the compound process

$$Workstations = Alice \interleave Bob \interleave Carl$$

We model the servers in the same style as the workstations with a server process for each server. A *Get* transaction process services a read request from a workstation and a *Set* transaction process services a write request. For simplicity, we make no attempt to provide serializable histories. Also, as in the original FRS approach, the servers make no security checks. Any process that has the fragment key may access a fragment. A *Get* transaction for sever *Deedee* looks like

$$Get_{Deedee}(w, f, s) = in?w.deedee.read.f.s \to out!deedee.w.f.s \to Skip$$

The *Get* transactions are indexed over the set of workstation names *Worksta*, the set of valid data fragments for the host $F_{host}$, and the set of read transaction identifiers *ReadID*.

$$Gets_{Deedee} = \mathop{|||}_{\substack{w \in Worksta \\ f \in F_{Deedee} \\ s \in ReadID}} Get_{Deedee}(w, f, s)$$

A *Set* transaction looks like

$$Set_{Deedee}(w, f, x, s) = in?w.h.write.x.f.s \rightarrow out!w.h.ack.s \rightarrow Skip$$

The *Set* transactions are indexed over the set of workstations $w \in Worksta$, the set of valid data fragments for host $f \in F_{host}$, the set of allowable data values $x \in Values$, and the set of write transaction identifiers $s \in WriteID$.

$$Sets_{Deedee} = \mathop{|||}_{\substack{w \in Worksta \\ f \in F_{Deedee} \\ x \in values \\ s \in WriteID}} Set_{Deedee}(w, f, x, s)$$

The interleaved Sets and Gets make up the server process, as in

$$Deedee = Gets_{Deedee} \; ||| \; Sets_{Deedee}$$

To complete our survivable system, we must connect the workstations and servers to the LAN. We rename the I/O events in the workstation and server processes to match their LAN connections using a renaming function. Renaming function *cable* is defined as

$$\begin{array}{lcll} cable(out!s.d.x) & = & in.s?s.d.x & \text{for } s, d \in Worksta \cup Serv \\ cable(out!s.d.x) & = & in.s?s.d.x & \text{for } s, d \in Worsta \cup Serv \\ cable(y) & = & y & \text{for all other events} \end{array}$$

The connection to the LAN process is by interface parallel

$$FRS = cable(Workstations) \; || \; cable(Servers) LAN$$

Figure 4 shows the process communication of the FRS solution.

We conclude with a brief analysis of this architecture according to our fault model. The FRS mechanism we have chosen implies the ten general security mechanism assertions. Our security policy brings in three more assertions. We will look at the ten general assertions, but defer identification of enforcement sets until we look at specific faults.

*Define and Separate Domains*

There are three kinds of domains in our example: workstation, LAN, and server. We examine each in turn.

A domain on the workstation is defined and separated by the workstation infrastructure and the architectural requirement that all of a workstation's storage (persistent and volatile) is erased at the end of each user session. Workstation

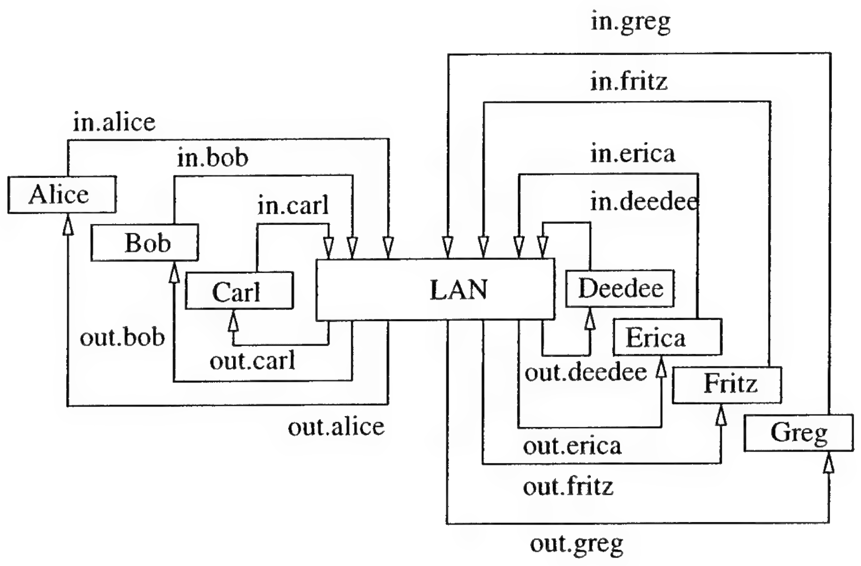

Figure 4: Process Algebra Communication Diagram

domains allow unlimited interpretation of instructions and data, on the workstation. A new domain is defined each time a session starts, by the erasure of a workstation's storage.

A domain on the LAN is defined and separated by the fragment key associated with a set of fragments. Any process possessing the fragment key will be able to properly concatenate the fragments (e.g. get $m_1$; $m_2$ instead of $m_2$; $m_1$ ) and decrypt them. Processes not holding the proper fragment key have their computation limited to cryptanalysis of all possible orderings of a complete set of encrypted fragments. A new domain is created each time a workstation requests a fragment key from the access controller and uses the key to fragment a copy of a database.

A domain on the server is defined and separated by both the fragment key and by scattering. The effect of the latter is that a domain contains only partial sets of encrypted fragments. New domains are created in a two step process: workstation fragmentation according to a fragment key and fragment scattering. Scattering is negotiated between the servers by a pseudo-random algorithm.

*Establish Initial Domains*

Initial domains are either set up by the infrastructure of a process, e.g. workstation process *Alice*, or by the access controller. In our solution, the access controller is protected by a separate FRS scheme that we will not model.

*Link External Actors with Domains*

This FRS architecture uses the workstation infrastructure to link external actors with workstation domains. The access controller is responsible for linking external actors with LAN and server domains. In an FRS scheme, workstations broadcast requests for fragments, in random order. Each server that holds a fragment always sends it to the requesting workstation, so the workstations and LAN do no linkage.

*Control Inter-domain Communication*

An FRS mechanism has no control over inter-domain communication, other than what is supplied by the workstation process infrastructure. That is, we expect either the infrastructure or the workstations themselves to protect fragment keys and not to broadcast assembled plaintext databases.

*Control Communication With the Environment*

Our example architecture shows a firewall that controls communication with the environment. A pure FRS mechanism does not control communication with its environment, since it is computationally unfeasible to reassemble and decrypt a meaningful set of fragments. We should mention that FRS also must trust the access controller not to give fragment keys to the environment.

*Detect and Handle Faults*

A common sense interpretation of the FRS mechanism would allow for auditing and intrusion detection by the infrastructure. Intrusion detection might be complicated by the opacity of the database fragments. Reported attempts to copy or modify fragments could not be linked to the database associated with the fragment.

## 4.3 A Survivable Fault

Now we can apply our fault model to see how a fragmentation-redundancy-scattering architecture survives a fault. We suppose a fault in one of the servers, say Deedee to be specific. The target set for this fault includes all of the servers, all of the workstations, and the authentication server. We suppose that the fault is a captured fault $fault_{Deedee}(P)$ with parameter process

$$P = Deedee \; ||| \; (|||_{f \in F_{Deedee}} \; out!bob.deedee.k.f \rightarrow Skip)$$

This fault replaces process *Deedee* with captured process described by the parameter of $fault_{Deedee}(P)$. We do not concern ourselves with which process may have captured server *Deedee.*

Notice that weak encryption generally prevents the captured server process $P$ from inspecting the fragments to see if they are $m$ fragments; that is, we cannot say **if**$f = m_1 \vee f = m_2$ **then** $out!bob.deedee.f$ because we cannot identify the fragments.

As it occurs, the fault has no potential to violate the security policy. However, if the fault propagates as a second captured mode fault $fault_{Bob}(P))$ resulting in the capture of workstation process *Bob* as parameter process

$$P = Bob \; ||| \; (\mu X.in?bob.deedee.k.f : Leak_{Deedee} \rightarrow X)$$

where $Leak_{Deedee}$ is the set $\{bob.deedee.k.f \mid f \in F_{Deedee}\}$ of all unauthorized messages that dump fragments to Bob. (Value $k$ is a nonce used to identify the messages used to leak the database.)

The damage to both processes is von Neuman damage, since their programs have been damaged. To remove the fault, the programs will have to be replaced.

At this point it looks like we have the potential to experience a security violation in this FRS system. That is, it looks like the assertion Bob may only access the $c$ database will now be violated. However, since Bob only gets $m_1$ fragments from server Deedee, and these are encrypted, he can't do much with them. In fact, the invalidation of this fault is the two self-protection assertions about von Neuman damage to to the original Bob and Deedee processes, since we have not given the $fault_{Bob}(P)$ process the ability to search the encrypted fragments and reassemble them, without the proper fragment key. The enforcement set for this assertion is $\{Alice, Carl, TAM\}$ since these three processes are responsible for encrypting the fragments of $m$ with a key that Bob does not know.

## 4.4 A Non-Survivable Fault

Our fault model is also useful for investigating faults that are not survivable. The FRS system we have proposed cannot survive a captured mode fault $fault_{Alice}(P)$ with parameter

$$P = CheckedRead_{Alice}(f, s);\ Leak(m)$$

where $Leak(m)$ is

$$\textbf{if}\ f_1 = m_1\ \textbf{then}\ out!alice.bob.k.f_1 \rightarrow out!alice.bob.k.f_2 \rightarrow Skip$$

The target of the fault $fault_{Alice}(P)$ is all of the workstations, the servers, and the authentication server. If this fault captures $CheckedRead_{Alice}(f, s)$ and also propagates a captured mode fault $fault_{Bob}(P')$ to workstation process Bob

$$P' = Bob\ |||\ (\mu X.in?bob.alice.k.f : \{m_1, m_2\} \rightarrow X)$$

The damage of these faults is also von Neuman, so the affected programs will need to be replaced with valid versions.

In this fault, the invalidation includes the application-specific assertion, "Bob may access just the c database" as well as the two von Neuman damage assertions. Bob does not get full access to the database, but only read access. It is also possible for workstations in our FRS architecture to leak fragment keys to one another which would grant full access, but we cannot model this fault with the simple authentication server process we have used here. We could model fragment keys, but the model would become too large for this paper.

## 5 Conclusions

Navy mission-critical systems have significant survivability requirements and unavoidable complexity. Systems that process national-security information are likely to be targets of sophisticated attacks. Unavoidable complexity may lead to unexpected accidental faults. A fault model that supports precise analysis of mission impact, response, and restoration of services is necessary for the design process. A precise fault model is also necessary for assurance and certification of survivability in Navy systems.

The four-tuple fault model (target, mode, damage, invalidation) gives us a way to describe the propagation of faults, the change in behavior, the means of repair, and the impact on system features. In this paper failure modes are described using a process algebra. Any well-defined model of computation may be used to describe failure modes. Examples of such models would be automata, grammars, and Petri nets.

The four-tuple fault model can be used to define taxonomies. One or more of the tuple components can be treated as a variable and the other components held constant. For example, we could suppose a tuple

$$(X, Y, \textit{von Neuman}, \{\textit{“Bob may access just the c database”}\})$$

to create a taxonomy of faults that allowed unauthorized access for Bob, in our FRS architecture. A carefully designed taxonomy could be used to define a flaw space. This flaw space would be useful in making design choices or trade-off studies. The precise definition of a flaw space depends on the model of computation used for the fault mode and the language used to specify the assertions in the invalidations.

Our model shows that survivability and security are related through ten basic assertions that must hold for any security mechanism. Survivable security mechanisms must withstand or tolerate faults that invalidate any of these assertions, as well as faults that invalidate application-specific assertions about confidentiality, integrity, or resource allocation. We can even go so far as to draw a useful distinction between survivability problems and fault-tolerance problems. Fault-tolerance solutions generally do not consider attacks that invalidate their assumptions or assertions. Proposed survivability solutions need to consider both the specified fault events and those fault events that invalidate their assumptions or assertions.

## References

[1] A. Avizenis, J. Laprie, and B. Randell. Fundamental concepts of dependability. In *Third Information Survivability Workshop*, Boston, MA, October 2000.

[2] D. Bernstine, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[3] Y. Deswarte, L. Blain, and J.-C. Fabre. Intrusion tolerance in distributed computing systems. In *IEEE Symposium on Research in Security and Privacy,* pages 110–121, 1991.

[4] C. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[5] C. Landwehr, B. Tretick, J. Carroll, and P. Anderson. A framework for evaluating computer architectures to support systems with security requirements, with applications. NRL Report 9088, Naval Research Laboratory, 5 November 1987.

[6] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *JACM*, 27(2):228–234, April 1980.

[7] J. Rushby and B. Randell. A distributed secure system. *IEEE Computer*, 16(7), 1983.